

Análise de Maus-Cheiros e Violação de Princípios de Bom Projeto

Grupo 18:

- Milena Baruc Rodrigues Moraes - 211062339
- Limírio Correia Guimarães - 200040201
- Lucas Alves Vilela - 211062141
- Pedro de Oliveira Campos Barbosa - 200026046

Introdução:

O desenvolvimento de software não se resume apenas à implementação de funcionalidades, mas também à qualidade do código produzido. Maus-cheiros de código são sinais de que algo pode estar errado na estrutura do código, tornando-o mais difícil de entender, manter e estender, identificar e corrigir esses problemas contribui para um código mais limpo, modular e sustentável a longo prazo.

Este trabalho foi confeccionado com base no livro [Refatoração: Aperfeiçoando o Design de Códigos Existentes](#) de Martin Fowler (Segunda edição -1999).

Parte - 1

Simplicidade

De acordo com o Dicio (2025), simplicidade é "qualidade daquilo que é simples; característica do que não é complexo; desprovido de complicação: a simplicidade dos manuais de instrução".

No nosso contexto, pode-se inferir que o código deve ser claro e direto, evitando complexidade desnecessária.

No capítulo 3 de seu livro, Fowler apresenta o tópico "Nome Misterioso", no qual afirma:

"Muitas vezes, as pessoas têm medo de renomear itens, achando que não vale a pena se dar ao trabalho, mas um bom nome pode fazer você economizar horas que seriam gastas quebrando a cabeça por falta de compreensão, no futuro."

Essa afirmação evidencia que uma prática simples, como nomear corretamente uma função ou variável, pode aumentar significativamente a manutenção do código, evitando também comentários desnecessários, o que mantém o arquivo mais limpo e simples.

Ainda no capítulo 3, Fowler apresenta o tópico "Código Duplicado", no qual afirma:

"Se você vir a mesma estrutura de código em mais de um lugar, pode ter certeza de que seu programa será melhor se encontrar uma forma de unificar

essas estruturas. Uma duplicação significa que, sempre que ler essas cópias, você terá de fazer isso cuidadosamente para ver se há alguma diferença.”

Essa afirmação destaca que a duplicação de código pode gerar inconsistências e aumentar o esforço de manutenção, pois qualquer alteração precisará ser replicada em múltiplos locais. Além disso, um código duplicado eleva a complexidade desnecessariamente, dificultando a leitura e a evolução.

Elegância

De acordo com o Dicio (2025), elegância é *"qualidade do que é elegante, distinto, refinado; distinção, refinamento"*.

No nosso contexto, pode-se inferir que o código deve ser bem estruturado e organizado. A elegância está na clareza e na facilidade de leitura, tornando o código mais compreensível e menos propenso a erros.

No capítulo 3 de seu livro, Fowler apresenta o tópico “Inveja de Recursos”, no qual afirma:

“Inveja de Recursos ocorre quando uma função em um módulo gasta mais tempo se comunicando com funções ou dados de outro módulo do que com o próprio módulo. Perdemos conta das vezes que vimos uma função chamar meia dúzia de métodos getter em outro objeto para calcular algum valor. Felizmente a cura para esse caso é óbvia: a função claramente quer estar com os dados, portanto use Mover Função (Move Function) para levá-la para lá.”

Observando essa afirmação, podemos concluir que a elegância do código está diretamente relacionada à sua coesão. Quando uma função depende excessivamente de outro módulo para acessar dados e realizar operações, isso indica um design frágil, dificultando a manutenção e a compreensão do sistema. Ao mover a função para o módulo ao qual faz mais sentido ela pertencer, evitamos o acoplamento desnecessário.

Modularidade

De acordo com o Dicio (2025), modular significa *"planejar ou construir em módulos ou partes independentes"*. Dentro do nosso contexto, pode-se inferir que o código deve ser dividido em módulos bem definidos.

No capítulo 3, Fowler apresenta o tópico “Alteração divergente”, onde ele explica que a Alteração Divergente ocorre quando um módulo é alterado por diferentes razões ou contextos, o que pode dificultar a manutenção e aumentar a complexidade. Fowler sugere que, para reduzir esse tipo de problema, devemos separar contextos diferentes em módulos independentes, para que cada um possa ser alterado de forma isolada. Esse conceito se associa ao princípio de modularidade, que sugere que o código deve ser dividido em módulos bem definidos e independentes, de forma que cada módulo tenha uma única responsabilidade.

Ainda no capítulo 3, Fowler apresenta o tópico “Cirurgia com rifle”, no qual afirma:

“Uma cirurgia com rifle é parecida com uma alteração divergente, porém é o oposto. Você sentirá o seu odor quando, sempre que fizer uma alteração, tiver de fazer várias modificações pequenas em várias classes diferentes. Quando as alterações estão por toda parte, elas são difíceis de encontrar, e será fácil se esquecer de uma modificação importante.”

Observando essa afirmação, podemos inferir que quando o código é bem modularizado, com cada módulo ou classe responsável por um único contexto ou funcionalidade, as alterações podem ser localizadas e isoladas dentro de módulos específicos. Isso reduz a necessidade de alterar múltiplos módulos simultaneamente, o que ajuda a manter o código mais organizado e facilita a manutenção.

Boas Interfaces

Ao pesquisar por “boas interfaces” no google, é apresentado uma visão geral criada por IA que define a pesquisa como:

“Uma boa interface é aquela que permite ao utilizador interagir com uma máquina de forma confortável e intuitiva, sem ter de recorrer a um manual de instruções.”

Dentro do nosso contexto esta definição está perfeitamente ajustada.

No capítulo 3, Fowler apresenta o tópico “Agrupamento de dado ” onde ele explica que dados que frequentemente aparecem juntos em várias partes do código, como campos em classes ou parâmetros em métodos, deveriam ser agrupados de forma mais coesa. O primeiro passo é identificar esses conjuntos de dados e transformá-los em um objeto. Isso ajuda a reduzir a complexidade das chamadas de métodos, tornando o código mais legível e fácil de manter. Ao agrupar dados relacionados em uma classe, tem a chance de melhorar ainda mais sua estrutura, criando um objeto mais completo e, assim, preparando o terreno para a reutilização de comportamentos e redução de duplicação.

Essa prática de agrupamento de dados e a criação de objetos coesos se relaciona diretamente ao conceito de boas interfaces de várias maneiras. Assim como uma boa interface permite que o usuário interaja de forma intuitiva e sem a necessidade de instruções complicadas, um código bem estruturado, com dados agrupados de forma lógica e organizada, facilita a interação dos desenvolvedores com o sistema.

Ao agrupar dados relacionados em objetos e reduzir a complexidade nas assinaturas de métodos, você cria um sistema em que a interface do código se torna mais limpa e intuitiva.

Extensibilidade

De acordo com o Dicio (2025), extensibilidade é “*qualidade do que é extensível, que pode ser estendido ou ampliado*”. Dentro do nosso contexto, pode-se inferir que o código deve permitir fácil adição de funcionalidades sem impactar o sistema.

No capítulo 3, Fowler apresenta o tópico “Switches repetidos”, onde ele explica que, ao encontrar a mesma lógica condicional (como instruções switch ou if/else) repetida em vários lugares, há o risco de tornar o código difícil de manter e expandir. Sempre que uma nova cláusula for adicionada, todos esses pontos de código precisarão ser atualizados, aumentando o risco de erros e tornando o processo mais trabalhoso. Fowler sugere que, para evitar esse problema, o polimorfismo pode ser utilizado como uma solução elegante, permitindo que a lógica condicional seja abstraída de maneira que facilite futuras extensões sem a necessidade de modificar vários pontos do código.

Isso se conecta diretamente à extensibilidade do sistema. Quando o código é bem projetado, utilizando princípios como o polimorfismo para eliminar repetições desnecessárias de lógica condicional, torna-se mais fácil adicionar novas funcionalidades sem alterar a estrutura existente.

Evitar duplicação

De acordo com o Dicio (2025), duplicação é "ato ou efeito de duplicar; cópia exata de algo; reprodução". Sabendo disso e tendo em conta o conceito de "evitar", podemos concluir em nosso contexto que devemos evitar trechos de código repetitivos, pois eles podem aumentar a complexidade e dificultar a manutenção do sistema.

No capítulo 3, Fowler apresenta o tópico "Código duplicado", que se trata exatamente da repetição de códigos, e ele afirma:

"Se você vir a mesma estrutura de código em mais de um lugar, pode ter certeza de que seu programa será melhor se encontrar uma forma de unificar essas estruturas. Uma duplicação significa que, sempre que ler essas cópias, você terá de fazer isso cuidadosamente para ver se há alguma diferença. Se houver necessidade de alterar o código duplicado, você deverá localizar e considerar cada duplicação."

Essa duplicação pode ser problemática, pois, ao necessitar de alterações, o programador precisará identificar todas as instâncias do código duplicado, o que aumenta a chance de erros e torna a manutenção mais trabalhosa.

Portabilidade

De acordo com o Dicio (2025), portabilidade é "qualidade do que é portátil, que pode ser transportado ou transferido de um lugar para outro com facilidade". Podemos inferir que, em nosso contexto, a portabilidade se refere ao código, que deve ser fácil de adaptar para diferentes ambientes.

No capítulo 3, Fowler apresenta o tópico "Dados Globais". Ele explica que dados globais, como variáveis globais ou singletons, podem ser modificados em qualquer ponto do código, dificultando o rastreamento de onde ocorreu a alteração. Isso pode gerar bugs difíceis de diagnosticar. A recomendação é encapsular esses dados para controlar melhor o acesso e limitar seu escopo, mantendo-os dentro de uma classe ou módulo específico.

Portanto, é importante minimizar o uso de dados globais e encapsulá-los sempre que possível, permitindo uma melhor adaptação e controle do código, o que favorece a portabilidade.

O código deve ser idiomático e bem documentado.

Ao observar toda a estrada que passamos até chegar neste tópico, podemos concluir que esse princípio, Código Idiomático e Bem Documentado, está intrinsecamente relacionado à clareza e manutenção do código. A ideia é escrever código que não apenas seja funcional, mas também compreensível e facilmente adaptável por outros desenvolvedores. Isso significa que o código deve seguir as convenções e padrões estabelecidos pela linguagem e pela equipe de desenvolvimento, tornando-o "idiomático", ou seja, escrito da maneira mais natural e esperada dentro do atual contexto.

Além disso, um código bem documentado não só facilita a compreensão imediata do que o código faz, mas também assegura que outros desenvolvedores possam fazer alterações ou adicionar novas funcionalidades de maneira mais simples e sem introduzir erros.

Podemos relacionar este tópico ao tópico “Nome Misterioso”, que já foi citado anteriormente aqui.

Parte - 2

1. Código Duplicado

- **Princípio Violado:** Código duplicado.

- **O que define este mau-cheiro:** Este mau-cheiro acontece quando uma mesma estrutura de código aparece em mais de um lugar, aumentando a complexidade e dificultando a manutenção do código.

- **Soluções comuns:**

- **Extraí função**, para transformar este trecho repetido em uma função reutilizável, **Deslocar instruções**

- Reorganizar o código agrupando estes trechos semelhantes para facilitar o processo de extração

- **Subir método** movimentando o código duplicado para a classe-base do código.

- **Mau-cheiro no código:** duplicação de código presente nos métodos **cadastrarDeducaoIntegral** e **cadastrarDependente**, onde um trecho de código de adicionar elementos a um array se repete em ambos os métodos.

```
1 public void cadastrarDependente(String nome, String parentesco) {
2     // adicionar dependente
3     String[] temp = new String[nomesDependentes.length + 1];
4     for (int i=0; i<nomesDependentes.length; i++) {
5         temp[i] = nomesDependentes[i];
6     }
7     temp[nomesDependentes.length] = nome;
8     nomesDependentes = temp;
9
10    String[] temp2 = new String[parentescosDependentes.length + 1];
11    for (int i=0; i<parentescosDependentes.length; i++) {
12        temp2[i] = parentescosDependentes[i];
13    }
14    temp2[parentescosDependentes.length] = parentesco;
15    parentescosDependentes = temp2;
16
17    numDependentes++;
18 }
```

Trecho do código do IRPF.java - linhas 42 a 59

```

1 public void cadastrarDeducaoIntegral(String nome, float valorDeducao) {
2     String temp[] = new String[nomesDeduccoes.length + 1];
3     for (int i=0; i<nomesDeduccoes.length; i++) {
4         temp[i] = nomesDeduccoes[i];
5     }
6     temp[nomesDeduccoes.length] = nome;
7     nomesDeduccoes = temp;
8
9     float temp2[] = new float[valoresDeduccoes.length + 1];
10    for (int i=0; i<valoresDeduccoes.length; i++) {
11        temp2[i] = valoresDeduccoes[i];
12    }
13    temp2[valoresDeduccoes.length] = valorDeducao;
14    valoresDeduccoes = temp2;
15 }

```

Trecho do código do IRPF.java - linhas 164 a 178

- **Refatoração: Extract Function** para extrair o trecho de código responsável por adicionar os elementos a um array.

2. Função Longa

- **Princípios Violados:** Simplicidade e Modularidade.

- **O que define este mau-cheiro:** Este mau-cheiro ocorre quando uma função ou método é muito grande e realiza várias tarefas ou contém muitas linhas de código, o que o torna difícil de entender, manter e testar.

- **Soluções comuns:**

- **Extrair Função (Extract Function):** Dividir a função em partes menores, criando novas funções para trechos de código que realizam tarefas específicas.

- **Decompor Condicional (Decompose Conditional):** Simplificar condicionais complexas extraíndo-as em funções separadas.

- **Substituir Variável Temporária por Consulta (Replace Temp with Query):** Eliminar variáveis temporárias desnecessárias que dificultam a extração de código.

- **Dividir Laço (Split Loop):** Separar laços que realizam múltiplas tarefas em funções distintas.

- **Mau-cheiro no código:** O método **calcularImpostos** na classe **CalculoImposto** é um exemplo de função longa, contendo várias condicionais e cálculos complexos em um único bloco de código o que o torna difícil de entender e desnecessariamente extenso.

```

1 public float calcularImpostos() {
2     float[] faixas = {2259.20f, 2826.65f, 3751.05f, 4664.68f};
3     float[] aliquotas = {0.0f, 0.075f, 0.15f, 0.225f, 0.275f};
4     float imposto = 0.0f;
5     float aux;
6
7     if (baseDeCalculo <= faixas[0]) {
8         return 0.0f;
9     }
10
11     if (baseDeCalculo > faixas[0] && baseDeCalculo <= faixas[1]) {
12         aux = baseDeCalculo - faixas[0];
13         return aux * aliquotas[1];
14     }
15
16     imposto += 42.59f;
17
18     if (baseDeCalculo > faixas[1] && baseDeCalculo <= faixas[2]) {
19         aux = baseDeCalculo - faixas[1];
20         return imposto + (aux * aliquotas[2]);
21     }
22
23     imposto += 138.66f;
24
25     if (baseDeCalculo > faixas[2] && baseDeCalculo <= faixas[3]) {
26         aux = baseDeCalculo - faixas[2];
27         return imposto + (aux * aliquotas[3]);
28     }
29
30     imposto += 205.57f;
31
32     aux = baseDeCalculo - faixas[3];
33     return imposto + (aux * aliquotas[4]);
34 }

```

Trecho do código do IRPF.java - linhas 303 a 336

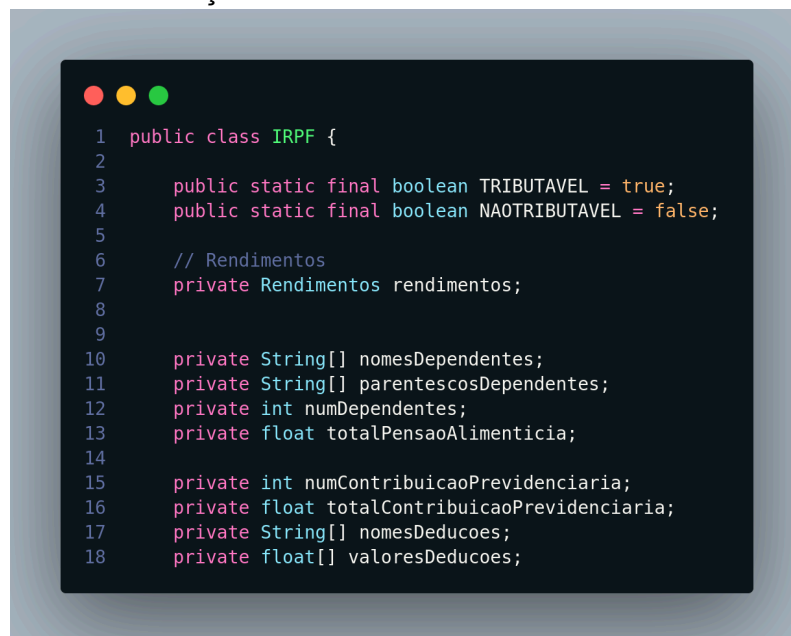
- Refatoração:

- Aplicar **Extrair Função** para dividir o método em partes menores, como extrair o cálculo de cada faixa de imposto em funções separadas.
- Usar o **Decompôr Condicional** para simplificar as condicionais presentes no método.
- Substituir a Variável Temporária 'aux' por uma consulta para reduzir o uso de variáveis temporárias e facilitar a extração de trechos de código.

3. Classe Grande

- **Princípio Violado:** Modularidade e Responsabilidade Única.
- **O que define este mau-cheiro:** Este mau-cheiro ocorre quando uma classe tenta fazer muitas coisas ao mesmo tempo, acumulando muitos campos e métodos. Classes grandes são difíceis de entender, manter e testar, pois misturam várias responsabilidades em um único componente. Além disso, elas frequentemente levam à duplicação de código e a um design pouco coeso.
- **Soluções comuns:**

- **Extrair Classe (Extract Class):** Dividir a classe em classes menores, agrupando campos e métodos relacionados em componentes separados.
- **Extrair Superclasse (Extract Superclass):** Criar uma superclasse para compartilhar funções e trechos de código comuns entre classes relacionadas.
- **Substituir Código de Tipos por Subclasses (Replace Type Code with Subclasses):** Usar subclasses para representar diferentes tipos ou funções e trechos de código, em vez de usar códigos de tipos dentro de uma única classe.
- **Eliminar Redundância:** Simplificar métodos longos e duplicados, extraíndo trechos de código repetidos e os transformando em métodos menores e reutilizáveis.
- **Mau-cheiro no código:** A classe IRPF é um exemplo de classe grande que tem muitas responsabilidades, como por exemplo gerenciar dependentes, deduções, contribuições previdenciárias, pensão alimentícia, etc. Isso a torna muito complexa e difícil de fazer manutenção.



```

1  public class IRPF {
2
3      public static final boolean TRIBUTAVEL = true;
4      public static final boolean NAOTRIBUTAVEL = false;
5
6      // Rendimentos
7      private Rendimentos rendimentos;
8
9
10     private String[] nomesDependentes;
11     private String[] parentescosDependentes;
12     private int numDependentes;
13     private float totalPensaoAlimenticia;
14
15     private int numContribuicaoPrevidenciaria;
16     private float totalContribuicaoPrevidenciaria;
17     private String[] nomesDeducoes;
18     private float[] valoresDeducoes;

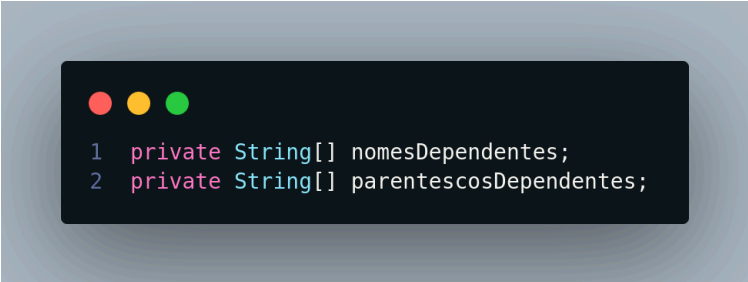
```

Trecho do código IRPF.java - linhas 5 a 22

- **Refatoração:**
 - Aplicar **Extrair Classe** para separar as responsabilidades da classe **IRPF** em classes menores, como:
 - Uma classe para gerenciar dependentes.
 - Uma classe para gerenciar deduções.
 - Uma classe para gerenciar contribuições previdenciárias.
 - Uma classe para gerenciar pensão alimentícia.
 - Usar **Extrair Superclasse** se houver funções e trechos de código comuns entre as novas classes.
 - Considerar **Substituir Código de Tipos por Subclasses** se houver lógicas específicas para diferentes tipos de deduções ou contribuições.

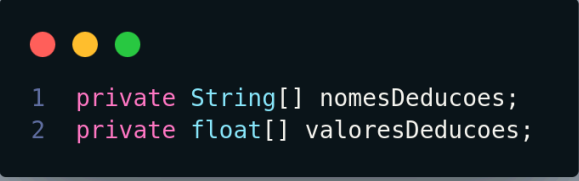
4. Obsessão por Primitivos

- **Princípio Violado:** Elegância e Abstração.
- **O que define este mau-cheiro:** Este mau-cheiro ocorre quando o código faz uso excessivo de tipos primitivos (como int, float, String) para representar conceitos complexos do domínio da aplicação. Isso pode levar a cálculos incorretos, falta de clareza e dificuldade de manutenção, especialmente quando os primitivos não capturam a semântica completa do que estão representando (por exemplo, moedas, datas, ou números de telefone).
- **Soluções comuns:**
 - **Substituir Primitivo por Objeto** (Replace Primitive with Object): Criar classes ou objetos para representar conceitos complexos, em vez de usar tipos primitivos.
 - **Extraí Classe** (Extract Class): Agrupar primitivos relacionados em uma classe, especialmente quando eles aparecem juntos frequentemente.
 - **Introduzir Objeto de Parâmetros** (Introduce Parameter Object): Substituir listas longas de parâmetros primitivos por um objeto que encapsula esses valores.
 - **Substituir Código de Tipos por Subclasses** (Replace Type Code with Subclasses): Usar subclasses para representar diferentes tipos ou funções e trechos de código, em vez de códigos de tipos primitivos.
 - **Substituir Condicional por Polimorfismo** (Replace Conditional with Polymorphism): Eliminar condicionais baseadas em tipos primitivos usando polimorfismo.
- **Mau-cheiro no código:** O uso de arrays primitivos (String[], float[]) para armazenar dependentes e deduções na classe IRPF é um exemplo de obsessão por primitivos. Esses arrays poderiam ser substituídos por objetos que encapsulam melhor a lógica e a semântica dos dados



```
1 private String[] nomesDependentes;  
2 private String[] parentescosDependentes;
```

Trecho do código IRPF.java - linhas 14 a 15



```
1 private String[] nomesDeduccoes;  
2 private float[] valoresDeduccoes;
```

Trecho do código IRPF.java - linhas 21 a 22

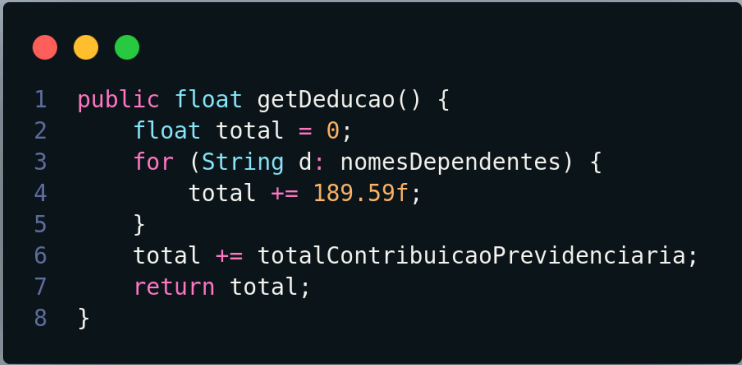
- **Refatoração:**

- Aplicar **Substituir Primitivo por Objeto** para criar classes como **Dependente** e **Dedução**, que encapsulam os dados, funções e trechos de código relacionados.
- Usar Extrair Classe para agrupar os arrays de dependentes e deduções em classes separadas, como **Dependentes** e **Deduccoes**.
- Introduzir **Objeto de Parâmetros** para simplificar métodos que recebem muitos parâmetros primitivos relacionados à dependentes ou deduções.

5. Feature Env

- **Princípio Violado:** Modularidade.
- **O que define o Mau-Cheiro**

No método *getDeducao()* da classe IRPF, temos a seguinte implementação:



```
1 public float getDeducao() {  
2     float total = 0;  
3     for (String d: nomesDependentes) {  
4         total += 189.59f;  
5     }  
6     total += totalContribuicaoPrevidenciaria;  
7     return total;  
8 }
```

Trecho do código IRPF.java - linhas 72 a 80

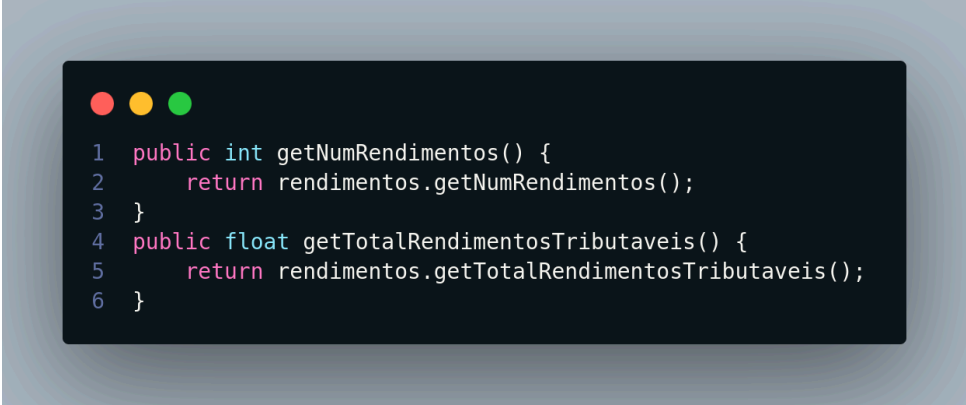
Esse trecho mostra que o método acessa diretamente os arrays *nomesDependentes* e a variável *totalContribuicaoPrevidenciaria*. Esse tipo de acesso direto mostra que a responsabilidade pelo cálculo das deduções está posicionada de forma incorreta e ambígua.

- **Operação de Refatoração**

Aplicar a refatoração *Move Method*, movendo esse método para uma nova classe dedicada ao gerenciamento das deduções, de modo que cada classe seja responsável por sua parte lógica.

6. Inappropriate Intimacy

- **Princípio Violado:** Boas Interfaces.
- **O que define o Mau-Cheiro:** A classe IRPF possui acesso direto aos métodos internos da classe Rendimentos, como demonstrado pelos seguintes trechos:



```
1 public int getNumRendimentos() {  
2     return rendimentos.getNumRendimentos();  
3 }  
4 public float getTotalRendimentosTributaveis() {  
5     return rendimentos.getTotalRendimentosTributaveis();  
6 }
```

Trecho do código IRPF.java - linhas 225 a 230

Ao acessar diretamente esses métodos, IRPF acaba conhecendo detalhes da implementação de Rendimentos, o que fere o princípio de boa abstração de interfaces.

- **Operação de Refatoração:** Encapsular o acesso a Rendimentos utilizando a técnica *Hide Delegate*, de forma que IRPF interaja com Rendimentos por meio de uma interface mais controlada e clara.

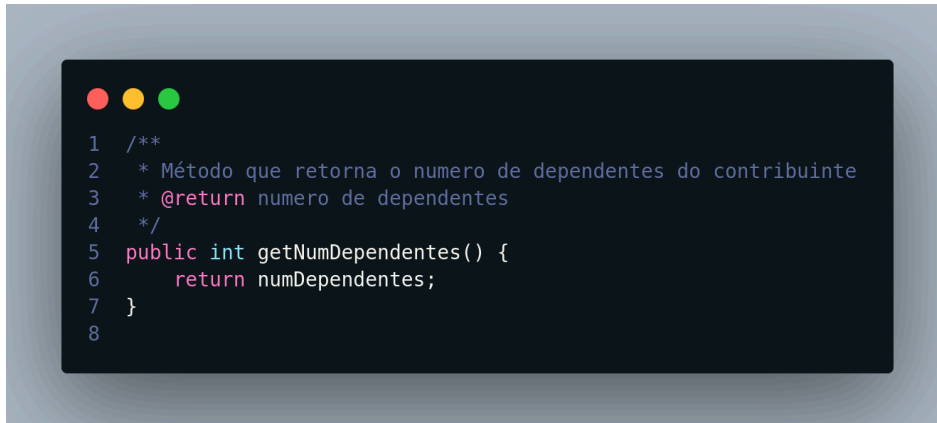
7. Shotgun Surgery

- **Princípio Violado:** Extensibilidade.
- **O que define o Mau-Cheiro :** Alterações nas regras de deduções ou no tratamento de dependentes impactam diversos métodos dentro da classe IRPF como por exemplo os métodos como ***cadastrarDependente()***, ***getDeducao()***, ***cadastrarPensaoAlimenticia()*** e ***getTotalPensaoAlimenticia()*** demonstram que a lógica relacionada a dependentes e deduções está espalhada. Esse tipo de dispersão obriga o desenvolvedor a modificar vários pontos do código para implementar uma única alteração, um tipo de problema conhecido como Shotgun Surgery.
- **Operação de Refatoração:** Centralizar a lógica relacionada a deduções e dependentes em classes separadas, utilizando refatorações como **Move Field** e **Move Method** para transferir os campos e métodos relevantes para onde eles realmente pertencem, facilitando a manutenção e evolução do código.

8. Comments

- **Princípio Violado:** Código Idiomático e Bem Documentado.

- **O que define o Mau-Cheiro:** O código contém diversos comentários explicativos, como os encontrados em:

A imagem mostra um trecho de código Java em um editor de texto com fundo escuro. O código está em português e contém comentários detalhados para uma função. As linhas são numeradas de 1 a 8.

```
1  /**
2   * Método que retorna o numero de dependentes do contribuinte
3   * @return numero de dependentes
4   */
5  public int getNumDependentes() {
6      return numDependentes;
7  }
8
```

Trecho do código IRPF.java - linhas 60 a 66

Esse excesso de comentários pode indicar que o código em si não é suficientemente claro e autoexplicativo, dependendo de explicações externas para ser compreendido.

- **Operação de Refatoração:** Em vez de depender de comentários para esclarecer a lógica, o ideal é melhorar os nomes dos métodos e das variáveis (aplicando refatorações como **Rename Method** e **Introduce Explaining Variable**), tornando o código mais intuitivo e facilitando sua compreensão sem a necessidade de explicações extensas em trechos de comentários.

Conclusão

Este trabalho teve como objetivo identificar e corrigir problemas no código, melhorando sua qualidade e aderência a boas práticas de desenvolvimento. Foram analisados maus-cheiros como código duplicado, funções longas e classes grandes, além de sugeridas refatorações para tornar o código mais claro, modular e sustentável.

A aplicação dessas técnicas facilita a manutenção e evolução do sistema, tornando-o mais eficiente e reduzindo a complexidade desnecessária. Além disso, reforça a importância de boas práticas, como evitar duplicação e manter a simplicidade, impactando positivamente a produtividade da equipe.

Por fim, a refatoração se mostra essencial para garantir um software mais robusto e preparado para futuras mudanças, promovendo um código mais legível e adaptável.

Referências Bibliográficas

DICIO. *Significado de simplicidade*. Disponível em: <https://www.dicio.com.br/simplicidade/>. Acesso em: 13 fev. 2025.

DICIO. *Significado de elegância*. Disponível em: <https://www.dicio.com.br/elegancia/>. Acesso em: 13 fev. 2025.

DICIO. *Significado de modular*. Disponível em: <https://www.dicio.com.br/modular/>. Acesso em: 13 fev. 2025.

DICIO. *Significado de extensibilidade*. Disponível em: <https://www.dicio.com.br/extensibilidade/>. Acesso em: 13 fev. 2025.

DICIO. *Significado de duplicação*. Disponível em: <https://www.dicio.com.br/duplicacao/>. Acesso em: 13 fev. 2025.

DICIO. *Significado de portabilidade*. Disponível em: <https://www.dicio.com.br/portabilidade/>. Acesso em: 13 fev. 2025.